INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Towards Practical Typechecking for Macro Tree Transducers

Alain Frisch — Haruo Hosoya

N° ????

Janvier 2007

Thème SYM

ISSN 0249-6399 ISRN INRIA/RR--????--FR+ENG

Towards Practical Typechecking for Macro Tree Transducers

Alain Frisch* , Haruo Hosoya†

Thème SYM — Systèmes symboliques
Projet Gallium

Rapport de recherche n° ???? — Janvier 2007 — 28 pages

Abstract: Macro tree transducers (mtt) are an important model that both covers many useful XML transformations and allows decidable exact typechecking. This paper reports our first step toward an implementation of mtt typechecker that has a practical efficiency. Our approach is to represent an input type obtained from a backward inference as an alternating tree automaton, in a style similar to Tozawa's XSLT0 typechecking. In this approach, typechecking reduces to checking emptiness of an alternating tree automaton. We propose several optimizations (Cartesian factorization, state partitioning) on the backward inference process in order to produce much smaller alternating tree automata than the naive algorithm, and we present our efficient algorithm for checking emptiness of alternating tree automata, where we exploit the explicit representation of alternation for local optimizations. Our preliminary experiments confirm that our algorithm has a practical performance that can typecheck simple transformations with respect to the full XHTML in a reasonable time.

Key-words: tree automata, tree transducers, exact typechecking, alternating automata

* INRIA, projet Gallium
† University of Tokyo

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Vers un typage praticable pour les macro transducteurs d'arbre

Résumé : Les macro transducteurs d'arbre (mtt) constituent un modèle important, dans la mesure où ils permettent de réaliser de nombreuses transformations XML et où ils admettent un typage exact décidable. Cet article rend compte d'une première étape en direction de l'implémentation d'un typeur pour les mtt efficace en pratique. Notre approche consiste à représenter le type d'entrée obtenu par inférence inverse sous la forme d'un automate d'arbre alternant, dans un style similaire à celui introduit par Tozawa pour le typage de XSLT0. Le problème de la vérification du bon typage du transducteur se réduit alors à celui du test de vide pour un automate d'arbre alternant. Nous proposons plusieurs optimisations (factorisation cartésienne, partionnement des états) pour le processus d'inférence inverse, avec l'objectif de produire des automates alternants significativement plus petits qu'avec l'algorithme naïf. Nous décrivons également un algorithme efficace pour le test de vide pour un automate d'arbre alternant, dans lequel nous exploitons la représentation explicite de l'alternation pour permettre des optimisations locales. Nos expériences préliminaires confirment que notre algorithme atteint des performances suffisantes pour typer des transformations par rapport à la DTD XHTML complète, en un temps raisonnable.

Mots-clés : automates d'arbre, transducteurs d'arbre, typage exact, automates alternants

1 Introduction

Static typechecking for XML transformations is an important problem that has expectedly a significant impact on real-world XML developments. To this end, several research groups have made efforts in building typed XML programming languages [8, 3] with much influence from the tradition of typed functional languages [2, 10]. While this line of work has successfully treated general, Turing-complete languages, its approximative nature has resulted in an even trivial transformation like the identity function to fail to typecheck unless a large amount of code duplicates and type annotations are introduced [7]. Such situation has led us to pay attention to completely different approaches that have no such deficiency, among which *exact typechecking* has emergingly become promising. The exact typechecking approach has extensively been investigated for years [12, 20, 16, 23, 26, 24, 11, 15, 1, 13, 18, 14], in which *macro tree transducers* (mtt) have been one of the most important models since they allow decidable exact typechecking [5], yet cover many useful XML transformations [5, 11, 4, 19]. Unfortunately, these studies are mainly theoretical and their practicality has never been clear except for some small cases [23, 26].

This paper reports our first step toward a *practical* implementation of typechecker for mtts. As a basic part, we follow an already-established scheme called backward inference, which computes the preimage of the output type for the subject transformation and then checks it against the given input type. This is because, as known well, the more obvious, forward inference does not work since the image of the input type is not always a regular tree language in general. Our proposal is, on top of this scheme, to use a representation of the preimage by an *alternating tree automaton* [21], extending the idea used in Tozawa's typechecking for XSLT0 [23]. In this approach, typechecking reduces to checking emptiness of an alternating tree automaton.

Whereas normal tree automata use only disjunctions in the transition relation, alternating tree automata can use both disjunctions and conjunctions. This extra freedom permits a more compact representation (they can be exponentially more succinct than normal tree automata) and make them a good intermediate language to study optimizations. Having explicit representation of transitions as Boolean formulas allowed us to derive optimized versions of the rules for backward inference, such as Cartesian decomposition or state partitioning (Section 4.1). These optimizations allow our algorithm to scale to large types. We also use Boolean reasoning to derive an efficient emptiness algorithm for alternating tree automata (Section 4.2). For instance, this algorithm uses the following fact as an efficient shortcut: when considering a formula $\phi = \phi_1 \wedge \phi_2$, if ϕ_1 turns out to denote an empty set, then so is ϕ, and thus the algorithm doesn't even need to look at ϕ_2. Note that the exploited fact is immediately available in alternating tree automata, while it is not in normal tree automata.

We have made extensive experiments on our implementation. We have written several sizes of transformations and verified against the full XHTML automatically generated from its DTD (in reality, transformations are often small, but types that they work on are quite big in many cases; excellent statistical evidences are provided in [17].) The results show that, for this scale of transformations, our implementation has successfully completed typechecking in a reasonable time even with XHTML, which is considered to be quite large. We have also compared the performance of our implementation with Tozawa and Hagiya's [26] and confirmed that ours has comparable speed for their small examples that are used in their own experiments.

On the theoretical side, we have established an exact relationship with two major existing algorithms for mtt typechecking, a classical algorithm based on "function enumeration" [4] and an algorithm proposed by Maneth, Perst, and Seidl (MPS algorithm) [12]. Concretely, we have

proved that (1) the classical algorithm is identical to our algorithm followed by determinization of an alternating tree automaton, and that (2) MPS algorithm is identical to our algorithm followed by emptiness test of an alternating tree automaton. A particular implication is that our algorithm inherits one of useful properties of MPS algorithm: polynomial-time complexity under the restriction of a bounded number of copying [12] (mtt typechecking is in general exponential-time complete). The proofs appear in the appendix, however, since this paper is focused rather on the practical side.

Related work Numerous techniques for exact typechecking for XML transformations have been proposed. Many of these take their target languages from the tree transducer family. Those include techniques for macro tree transducers [12, 4], for macro forest transducers [20], for k-pebble tree transducers [16, 4], for subsets of XSLT [23, 26], for high-level tree transducers [24], and a tree transformation language TL [11]. Other techniques treat XML query languages in the select-construct style [15, 1, 13] or even simpler transformations [18, 14]. Most of the above mentioned work provides only theoretical results; the only exceptions are [23, 26], where some experimental results are shown though we have examined much bigger examples (in particular in the size of types).

Several algorithms in pragmatic approaches have been proposed to address high complexity problems related to XML typechecking. A top-down algorithm for inclusion test on tree automata has been developed and used in XDuce typechecker [9]; an improved version is proposed in [22]. A similar idea has been exploited in the work on CDuce on the emptiness check for alternating tree automata [6]; the emptiness check algorithm in our present work is strongly influenced by this. Tozawa and Hagiya have developed BDD-based algorithms for inclusion test on tree automata [25] and for satisfiability test on a certain logic related to XML typechecking [26].

Overview This paper is organized as follows. In Section 2, we recall the classical definitions of macro tree transducers (mtt), bottom-up tree automata (bta), and alternating tree automata (ata). In Section 3, we present the two components of our typechecking algorithm: backward type inference (which produces an ata from an mtt and a deterministic bta) and emptiness check for alternating tree automata. In Section 4, we revisit these two components from a practical point of view and we describe important optimizations and implementation techniques. In Section 5, we report the results of our experiments with our implementation of the typechecker for several XML transformations. In Section 6, we conclude this paper with our future direction. Appendix A is devoted to a precise comparison between our algorithm and the classical algorithm or the Maneth-Perst-Seidl algorithm for typechecking mtt. We show that each of these algorithms can be retrieved from ours by composing with a know algorithm. In Appendix B, we propose the notion of bounded-traversing alternating tree automata, which is a natural counterpart of syntactical bounded-copying mtts as proposed in [12]. We show in particular that this notion ensures that the emptiness check runs in polynomial time.

2 Preliminaries

2.1 Macro Tree Transducers

We assume an alphabet Σ where each symbol $a \in \Sigma$ is associated with its arity; often we write $a^{(n)}$ to denote a symbol a with arity n. We assume that there is a symbol ϵ with zero-arity.

Trees, ranged over by $v, w, \ldots$, are defined as follows.

$$v \quad ::= \quad a^{(n)}(v_1, \ldots, v_n)$$

We write ϵ for $\epsilon()$ and $\vec{v} = (v_1, \ldots, v_n)$ to represent a tuple of trees. Assume a set of variables, ranged over by $x, y, \ldots$. A *macro tree transducer* (mtt) $\mathcal{T}$ is a tuple (P, P_0, Π) where P is a finite set of procedures, $P_0 \subseteq P$ is a set of initial procedures, and Π is a set of (transformation) rules each of the form

$$p^{(k)}(a^{(n)}(x_1, \ldots, x_n), y_1, \ldots, y_k) \to e$$

where each y_i is called *(accumulating) parameter* and e is a (n, k)-expression. We will abbreviate the tuples $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_k)$ to $\vec{x}$ and $\vec{y}$. Note that each procedure is associated with its arity, i.e., the number of parameters; we write $p^{(k)}$ to denote a procedure p with arity k. An (n, k)-expression e is defined by the following grammar

$$e \quad ::= \quad a^{(m)}(e_1, \ldots, e_m) \mid p^{(l)}(x_h, e_1, \ldots, e_l) \mid y_j$$

where only y_j with $1 \leq j \leq k$ and x_h with $1 \leq h \leq n$ can appear as variables. We assume that each initial procedure has arity zero.

We describe the semantics of an mtt (P, P_0, Π) by a denotation function $[\![\cdot]\!]$. First, the semantics of a procedure $p^{(k)}$ takes a tree $a^{(n)}(v_1, \ldots, v_n)$ and parameters $\vec{w} = (w_1, \ldots, w_k)$ and returns the set of trees resulted from evaluating any of p's body expressions.

$$[\![p^{(k)}]\!](a^{(n)}(\vec{v}), \vec{w}) = \bigcup_{(p^{(k)}(a^{(n)}(\vec{x}), \vec{y}) \to e) \in \Pi} [\![e]\!](\vec{v}, \vec{w})$$

Then, the semantics of an (n, k)-expression e takes a current n-tuple $\vec{v} = (v_1, \ldots, v_n)$ of trees and a k-tuple of parameters $\vec{w} = (w_1, \ldots, w_k)$, and returns the set of trees resulted from the evaluation. It is defined as follows.

$$\begin{array}{lcl}
[\![a^{(m)}(e_1, \ldots, e_m)]\!](\vec{v}, \vec{w}) & = & \{a^{(m)}(v'_1, \ldots, v'_m) \mid v'_i \in [\![e_i]\!](\vec{v}, \vec{w}) \text{ for } i = 1, \ldots, m\} \\
[\![p^{(l)}(x_h, e_1, \ldots, e_l)]\!](\vec{v}, \vec{w}) & = & \{[\![p^{(l)}]\!](v_h, (w'_1, \ldots, w'_l)) \mid w'_j \in [\![e_j]\!](\vec{v}, \vec{w}) \text{ for } j = 1, \ldots, l\} \\
[\![y_j]\!](\vec{v}, \vec{w}) & = & \{w_j\}
\end{array}$$

A constructor expression $a^{(m)}(e_1, \ldots, e_m)$ evaluates each subexpression e_i and reconstructs a tree node with a and the results of these subexpressions. A procedure call $p(x_h, e_1, \ldots, e_l)$ evaluates the procedure p under the h-th subtree v_h, passing the results of $e_1, \ldots, e_l$ as parameters. A variable expression y_j simply results in the corresponding parameter's value w_j. Note that an mtt is allowed to inspect only the input tree and never a part of the output tree being constructed. Also, parameters only accumulate subtrees that will potentially become part of the output and never point to parts of the input.

The whole semantics of the mtt with respect to a given input tree v is defined by $\mathcal{T}(v) = \bigcup_{p_0 \in P_0} [\![p_0]\!](v)$. An mtt $\mathcal{T}$ is deterministic when $\mathcal{T}(v)$ has at most one element for any v; also, $\mathcal{T}$ is total when $\mathcal{T}(v)$ has at least one element for any v. We will also use the classical definition of images and preimages: $\mathcal{T}(V) = \bigcup_{v \in V} \mathcal{T}(v)$, $\mathcal{T}^{-1}(V') = \{v \mid \exists v' \in V'.v' \in \mathcal{T}(v)\}$.

2.2 Tree Automata and Alternation

A *(bottom-up) tree automaton* (bta) $\mathcal{M}$ is a tuple (Q, Q_F, Δ) where Q is a finite set of states, $Q_F \subseteq Q$ is a set of final states, and Δ is a set of (transition) rules each of the form $q \leftarrow$

An *alternating tree automaton* (ata) $\mathcal{A}$ is a tuple (Ξ, Ξ_0, Φ) where Ξ is a finite set of states, $\Xi_0 \subseteq \Xi$ is a set of initial state, and Φ is a function that maps each pair $(X, a^{(n)})$ of a state and an n-ary constructor to an n-formula, where n-formulas are defined by the following grammar.

$$\phi \quad ::= \quad \downarrow_i X \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \top \mid \bot$$

(with $1 \leq i \leq n$). In particular, note that a 0-ary formula evaluates naturally to a Boolean. Given an ata $\mathcal{A} = (\Xi, \Xi_0, \Phi)$, we define acceptance of a tree by a state. $\mathcal{A}$ accepts a tree $a^{(n)}(\vec{v})$ by a state X when $\vec{v} \vdash \Phi(X, a^{(n)})$ holds, where the judgment $\vec{v} \vdash \phi$ is defined inductively as follows:

- $\vec{v} \vdash \phi_1 \wedge \phi_2$ if $\vec{v} \vdash \phi_1$ and $\vec{v} \vdash \phi_2$.
- $\vec{v} \vdash \phi_1 \vee \phi_2$ if $\vec{v} \vdash \phi_1$ or $\vec{v} \vdash \phi_2$.
- $\vec{v} \vdash \top$.
- $\vec{v} \vdash \downarrow_i X$ if $\mathcal{A}$ accepts v_i by X.

That is, $\vec{v} \vdash \phi$ intuitively means that ϕ holds by interpreting each $\downarrow_i X$ as "v_i has type X." We write $[\![X]\!]$ for the set of trees accepted by a state X and $[\![\phi]\!] = \{\vec{v} \mid \vec{v} \vdash \phi\}$ for the set of n-tuples accepted by an n-formula ϕ. We write $\mathcal{L}(\mathcal{A}) = \bigcup_{X_0 \in \Xi_0} [\![X_0]\!]$ for the language accepted by the ata $\mathcal{A}$. Note that a bta $\mathcal{M} = (Q, Q_F, \Delta)$ can be seen as an ata with the same set of states and final states by defining the function Φ as $\Phi(q, a^{(n)}) = \bigvee_{(q \leftarrow a^{(n)}(\vec{q})) \in \Delta} \bigwedge_{i=1,..,n} \downarrow_i q_i$, and the definitions for the semantics of states and the language accepted by the automaton seen as a bta or an ata then coincide. We will use the notation $\simeq$ to represent semantical equivalence of pairs of states or pairs of formulas.

3 Typechecking

3.1 Backward inference

Given a dbta $\mathcal{M}_{\text{out}}$ ("output type"), a bta $\mathcal{M}_{\text{in}}$ ("input type"), and an mtt $\mathcal{T}$, the goal of typechecking is to verify that $\mathcal{T}(\mathcal{L}(\mathcal{M}_{\text{in}})) \subseteq \mathcal{L}(\mathcal{M}_{\text{out}})$. It is well known that $\mathcal{T}(\mathcal{L}(\mathcal{M}_{\text{in}}))$ is in general beyond regular tree languages and hence the forward inference approach (i.e., first calculate an automaton representing $\mathcal{T}(\mathcal{L}(\mathcal{M}_{\text{in}}))$ and check it to be included in $\mathcal{L}(\mathcal{M}_{\text{out}})$) does not work. Therefore an approach usually taken is the backward inference, which is based on the observation that $\mathcal{T}(\mathcal{L}(\mathcal{M}_{\text{in}})) \subseteq \mathcal{L}(\mathcal{M}_{\text{out}}) \iff \mathcal{L}(\mathcal{M}_{\text{in}}) \cap \mathcal{T}^{-1}(\mathcal{L}(\mathcal{M})) = \emptyset$, where $\mathcal{M}$ is

the complement automaton of $\mathcal{M}_{\text{out}}$. Intuitively, if the intersection $\mathcal{L}(\mathcal{M}_{\text{in}}) \cap \mathcal{T}^{-1}(\mathcal{L}(\mathcal{M}))$ is not empty, then it is possible to exhibit a tree v in this intersection. Since this tree satisfies that $v \in \mathcal{L}(\mathcal{M}_{\text{in}})$ and $\mathcal{T}(v) \not\subseteq \mathcal{L}(\mathcal{M}_{\text{out}})$, it means that there is a counter-example of the well-typedness of the mtt with respect to the given input and output types. Algorithmically, the approach consists of computing an automaton $\mathcal{A}$ representing $\mathcal{T}^{-1}(\mathcal{L}(\mathcal{M}))$ and then checking that $\mathcal{L}(\mathcal{M}_{\text{in}}) \cap \mathcal{L}(\mathcal{A}) = \emptyset$. Since the language $\mathcal{T}^{-1}(\mathcal{L}(\mathcal{M}))$ is regular and indeed such automata $\mathcal{A}$ can effectively be computed, the above disjointness is decidable.

The originality of our approach is to compute $\mathcal{A}$ as an alternating tree automaton. Let a dbta $\mathcal{M} = (Q, Q_F, \Delta)$ and an mtt $\mathcal{T} = (P, P_0, \Pi)$ be given. Here, note that the automaton $\mathcal{M}$, which denotes the complement of the output type $\mathcal{M}_{\text{out}}$, can be obtained from $\mathcal{M}_{\text{out}}$ in a linear time since $\mathcal{M}_{\text{out}}$ is deterministic. From $\mathcal{M}$ and $\mathcal{T}$, we build an ata $\mathcal{A} = (\Xi, \Xi_0, \Phi)$ where

$$\begin{array}{lcl} \Xi & = & \{\langle p^{(k)}, q, \vec{q}\rangle \mid p^{(k)} \in P,\ q \in Q, \vec{q} \in Q^k\} \\ \Xi_0 & = & \{\langle p_0, q\rangle \mid p_0 \in P_0,\ q \in Q_F\} \\ \Phi(\langle p^{(k)}, q, \vec{q}\rangle, a^{(n)}) & = & \displaystyle\bigvee_{(p^{(k)}(a^{(n)}(\vec{x}),\vec{y}) \to e) \in \Pi} \mathrm{Inf}(e, q, \vec{q}). \end{array}$$

Here, the function Inf is defined inductively as follows.

$$\begin{array}{lcl} \mathrm{Inf}(b^{(m)}(e_1, \ldots, e_m), q, \vec{q}) & = & \displaystyle\bigvee_{(q \leftarrow b^{(m)}(\vec{q'})) \in \Delta} \bigwedge_{j=1,..m} \mathrm{Inf}(e_j, q'_j, \vec{q}) \\ \mathrm{Inf}(p^{(l)}(x_h, e_1, \ldots, e_l), q, \vec{q}) & = & \displaystyle\bigvee_{\vec{q'} \in Q^l} \left(\downarrow_h \langle p^{(l)}, q, \vec{q'}\rangle \wedge \bigwedge_{j=1,..,l} \mathrm{Inf}(e_j, q'_j, \vec{q}) \right) \\ \mathrm{Inf}(y_j, q, \vec{q}) & = & \begin{cases} \top & (q = q_j) \\ \bot & (q \neq q_j) \end{cases} \end{array}$$

Let us explain why this algorithm works. Since a precise discussion is critical for understanding subsequent sections, we summarize our justification here as a formal proof.

Theorem 1 $\mathcal{L}(\mathcal{A}) = \mathcal{T}^{-1}(\mathcal{L}(\mathcal{M}))$.

PROOF: Intuitively, each state $\langle p, q, \vec{q}\rangle$ represents the set of trees v such that the procedure p may transform v to some tree u of type q, assuming that the parameters y_i are bound to trees w_i each of type q_i. Formally, we prove the following invariant

$$\forall v.\ \forall \vec{w} \in [\![\vec{q}]\!].\ v \in [\![\langle p^{(k)}, q, \vec{q}\rangle]\!] \iff [\![p^{(k)}]\!](v, \vec{w}) \cap [\![q]\!] \neq \emptyset \qquad (1)$$

where $\vec{w} \in [\![\vec{q}]\!]$ means $w_1 \in [\![q_1]\!], \ldots, w_k \in [\![q_k]\!]$. Note that this invariant implies that the right-hand side does not depend on the specific choice of the values w_i from the sets $[\![q_i]\!]$; this point will be crucial later. From this invariant, the initial states Ξ_0 represent the set of trees that we want and hence the result follows:

$$\begin{array}{lcl} \mathcal{L}(\mathcal{A}) & = & \bigcup\{[\![\langle p_0, q\rangle]\!] \mid p_0 \in P_0,\ q \in Q_F\} \\ & = & \{v \mid [\![p_0]\!](v) \cap [\![q]\!] \neq \emptyset,\ p_0 \in P_0,\ q \in Q_F\} \\ & = & \{v \mid \mathcal{T}(v) \cap \mathcal{L}(\mathcal{M}) \neq \emptyset\} \\ & = & \mathcal{T}^{-1}(\mathcal{L}(\mathcal{M})) \end{array}$$

The proof of the invariant (1) proceeds by induction on the structure of v. For the proof, we first need to consider an invariant that holds for the function Inf. Informally, $\mathrm{Inf}(e, q, \vec{q})$ infers

an n-formula representing the set of n-tuples $\vec{v}$ such that the expression e may transform $\vec{v}$ to some tree of type q, assuming that the parameters y_i are bound to trees w_i each of type q_i. Formally, we prove the following:

$$\forall \vec{v}.\ \forall \vec{w} \in [\![\vec{q}]\!].\ \vec{v} \in [\![\mathrm{Inf}(e, q, \vec{q})]\!] \iff [\![e]\!](\vec{v}, \vec{w}) \cap [\![q]\!] \neq \emptyset \tag{2}$$

Indeed, this implies the invariant (1). Let $v = a^{(n)}(\vec{v})$; for all $\vec{w} \in [\![\vec{q}]\!]$:

$$\begin{array}{rcl} v \in [\![\langle p^{(k)}, q, \vec{q}\rangle]\!] & \iff & \vec{v} \in [\![\Phi(\langle p^{(k)}, q, \vec{q}\rangle, a^{(n)})]\!] \\ & \iff & \exists (p^{(k)}(a^{(n)}(\vec{x}), \vec{y}) \to e) \in \Pi.\ \vec{v} \in [\![\mathrm{Inf}(e, q, \vec{q})]\!] \\ & \overset{by(2)}{\iff} & \exists (p^{(k)}(a^{(n)}(\vec{x}), \vec{y}) \to e) \in \Pi.\ [\![e]\!](\vec{v}, \vec{w}) \cap [\![q]\!] \neq \emptyset \\ & \iff & [\![p]\!](v, \vec{w}) \cap [\![q]\!] \neq \emptyset \end{array}$$

The invariant (2) is in turn proved by induction on the structure of e.

Case $e = b^{(m)}(e_1, \ldots, e_m)$**.** In order for a tree u of type q to be produced from the constructor expression, first, there must be a transition $q \leftarrow b^{(m)}(\vec{q'}) \in \Delta$. In addition, u's each subtree must have type q'_i and must be produced from the corresponding subexpression e_i. For the latter condition, we can use the induction hypothesis for (2). Formally, for all $\vec{w} \in [\![\vec{q}]\!]$:

$$\begin{array}{rcl} \vec{v} \in [\![\mathrm{Inf}(e, q, \vec{q})]\!] & \iff & \vec{v} \in [\![\bigvee_{q \to b^{(m)}(\vec{q'}) \in \Delta} \bigwedge_{j=1,\ldots,m} \mathrm{Inf}(e_j, q'_j, \vec{q})]\!] \\ & \iff & \exists (q \leftarrow b^{(m)}(\vec{q'})) \in \Delta.\ \forall j = 1, \ldots, m.\ \vec{v} \in [\![\mathrm{Inf}(e_j, q'_j, \vec{q})]\!] \\ & \overset{\text{byI.H.for}(2)}{\iff} & \exists (q \leftarrow b^{(m)}(\vec{q'})) \in \Delta.\ \forall j = 1, \ldots, m.\ [\![e_j]\!](\vec{v}, \vec{w}) \cap [\![q'_j]\!] \neq \emptyset \\ & \iff & [\![e]\!](\vec{v}, \vec{w}) \cap [\![q]\!] \neq \emptyset \end{array}$$

Case $e = p^{(l)}(x_h, e_1, \ldots, e_l)$**.** In order for a tree u of type q to be produced from the procedure call, first, a tree w'_j of some type q'_j must be yielded from each parameter expression e_j. In addition, the h-th input tree must have type $\langle p^{(l)}, q, (q'_1, \ldots, q'_l)\rangle$ since the result tree u must be produced by the procedure $p^{(l)}$ from the h-th input tree with parameters $w'_1, \ldots, w'_l$ of types $q'_1, \ldots, q'_l$. We can use the induction hypothesis for (2) for the former condition and that for (1) for the latter condition. Formally, for all $\vec{w} \in [\![\vec{q}]\!]$:

$$\begin{array}{rcl} \vec{v} \in [\![\mathrm{Inf}(e, q, \vec{q})]\!] & \iff & \vec{v} \in [\![\bigvee_{\vec{q'} \in Q^l} \downarrow_h \langle p, q, \vec{q'}\rangle \wedge \bigwedge_{j=1,\ldots,l} \mathrm{Inf}(e_j, q'_j, \vec{q})]\!] \\ & \iff & \exists \vec{q'} \in Q^l.\ v_h \in [\![\langle p, q, \vec{q'}\rangle]\!] \wedge \forall j = 1, \ldots, l.\ \vec{v} \in [\![\mathrm{Inf}(e_j, q'_j, \vec{q})]\!] \\ & \overset{\text{byI.H.for}(2)}{\iff} & \exists \vec{q'} \in Q^l.\ v_h \in [\![\langle p, q, \vec{q'}\rangle]\!] \wedge \forall j = 1, \ldots, l.\ [\![e_j]\!](\vec{v}, \vec{w}) \cap [\![q'_j]\!] \neq \emptyset \\ & \iff & \exists \vec{q'} \in Q^l.\ \ v_h \in [\![\langle p, q, \vec{q'}\rangle]\!] \qquad (3) \\ & & \qquad \wedge\ \exists \vec{w'}.\ \forall j = 1, \ldots, l.\ w'_j \in [\![e_j]\!](\vec{v}, \vec{w})\ \wedge\ w'_j \in [\![q'_j]\!] \end{array}$$

We can show that the last condition holds iff

$$\exists \vec{w'}.\ \ [\![p^{(l)}]\!](v_h, \vec{w'}) \cap [\![q]\!] \neq \emptyset\ \wedge\ \forall j = 1, \ldots, l.\ w'_j \in [\![e_j]\!](\vec{v}, \vec{w}) \tag{4}$$

which is equivalent to $[\![p(x_h, e_1, \ldots, e_m)]\!] \cap [\![q]\!] \neq \emptyset$. Indeed, for the "only if" direction, we apply the induction hypothesis for (1) where we instantiate $\vec{w}$ with the specific $\vec{w'}$ in (3)—this is exactly the place that uses the fact that the quantification on $\vec{w}$ appears outside the " $\Longleftrightarrow$ " in (1)—and obtain the following:

$$\exists \vec{q'} \in Q^l.\ \exists \vec{w'}.\quad [\![p^{(l)}]\!](v_h, \vec{w'}) \cap [\![q]\!] \neq \emptyset \\ \wedge\ \forall j = 1, \ldots, l.\ w'_j \in [\![e_j]\!](\vec{v}, \vec{w})\ \wedge\ w'_j \in [\![q'_j]\!] \tag{5}$$

By dropping the condition $w'_j \in [\![q'_j]\!]$ (and the unused quantification on $\vec{q'}$), we obtain (4).

For the "if" direction, since that the automaton $\mathcal{M}$ is complete, i.e., there is in general a state q for any value w such that $w \in [\![q]\!]$, we obtain (5) from (4). Then, the induction hypothesis for (1) yields (3).

Case $e = y_j$**.** In order for a tree of type q to be produced from the variable expression, y_j must have type q. Formally, first note that $\vec{v} \in [\![\mathrm{Inf}(e, q, \vec{q})]\!] \iff q = q_j$, for any $\vec{v}$. Note also that, since $\mathcal{M}$ is deterministic bottom-up, all the states are pair-wise disjoint: $[\![q]\!] \cap [\![q']\!] = \emptyset$ whenever $q \neq q'$. Therefore, for all $\vec{w} \in [\![\vec{q}]\!]$:

$$\begin{array}{rcl} \vec{v} \in [\![\mathrm{Inf}(e, q, \vec{q})]\!] & \Longleftrightarrow & q = q_j \\ & \Longleftrightarrow & w_j \in [\![q]\!] \\ & \Longleftrightarrow & [\![e]\!](\vec{v}, \vec{w}) \cap [\![q]\!] \neq \emptyset \end{array}$$

□

In the proof above, the case for variable expressions critically uses the determinism constraint. Indeed, the statement of the theorem does not necessarily hold if $\mathcal{M}$ is nondeterministic. For example, consider the nondeterministic bta $\mathcal{M}$ with the transition rules

$$q_0 \leftarrow b(q_1, q_2) \quad q_1 \leftarrow \epsilon \quad q_2 \leftarrow \epsilon$$

(q_0 is the initial state) and typecheck the mtt $\mathcal{T}$ with the transformation rules

$$\begin{array}{rcl} p_0(a(x_1)) & \to & p(x_1, \epsilon) \\ p(\epsilon, y_1) & \to & b(y_1, y_1) \end{array}$$

(p_0 is the initial procedure) with respect to the result type q_0. With this mtt, the input value $a(\epsilon)$ translates to $b(\epsilon, \epsilon)$, which is accepted by $\mathcal{M}$. However, our algorithm will infer an input type that denotes the empty set, which is incorrect. To see this more closely, consider inference on the body of p with the result type $q = q_0$ and the parameter type $\vec{q} = (q_1)$. The condition (2) does not hold since the only choice of $\vec{w} \in [\![\vec{q}]\!]$ is $\vec{w} = (\epsilon)$ and, in this case, the right hand side holds whereas the left hand side does not since $\mathrm{Inf}(b(y_1, y_1), q_0, (q_1)) = \mathrm{Inf}(y_1, q_1, (q_1)) \wedge \mathrm{Inf}(y_1, q_2, (q_1)) = \top \wedge \bot = \bot$. The same argument can be done with the parameter type $\vec{q} = (q_2)$. Now, in inference on the body of p_0 with the result type q_0, the call to p must have parameter type q_1 or q_2 since only these can accept ϵ. From the previous inference, we conclude that the input type inferred for the call is again the empty set type; so is the whole input type.

However, the variable case is the only that uses determinism. Therefore, if the mtt uses no parameter, i.e., is a simple, top-down tree transducer, then the same algorithm works for a non-deterministic output type.[1] Moreover, if the mtt $\mathcal{T}$ is deterministic and total, we have

[1] Completeness of the output type is not needed for our algorithm to work on top-down tree transducers. This is because the only place where we use completeness in the proof is the case for procedure calls, in which completeness is actually not necessary if there is no parameter.

$\overline{\mathcal{T}^{-1}(\mathcal{L}(\overline{\mathcal{M}_{\text{out}}}))} = \overline{\mathcal{T}^{-1}(\mathcal{L}(\mathcal{M}_{\text{out}}))}$. It suffices to check $\mathcal{L}(\mathcal{M}_{\text{in}}) \subseteq \mathcal{T}^{-1}(\mathcal{L}(\mathcal{M}_{\text{out}}))$ instead of $\mathcal{L}(\mathcal{M}_{\text{in}}) \cap \mathcal{T}^{-1}(\mathcal{L}(\overline{\mathcal{M}_{\text{out}}})) = \emptyset$. This could be advantageous since a direct conversion from an XML schema yields a non-deterministic automaton, and determinizing it has a potential blow-up (though this step is known to take only a reasonable time in practice) whereas inclusion can be tested more efficiently by using known clever algorithms that avoid a full materialization of a deterministic automaton [9, 22, 25]. Tozawa presents in his work [23] a backward inference algorithm based on alternating tree automata for deterministic forest transducers with no parameters where he exploits the above observation to obtain a simple algorithm.

Finally, it remains to check $\mathcal{L}(\mathcal{M}_{\text{in}}) \cap \mathcal{L}(A) = \emptyset$, for which we first calculate an ata $\mathcal{A}'$ representing $\mathcal{L}(\mathcal{M}_{\text{in}}) \cap \mathcal{L}(A)$ (this can easily be done since an ata can freely use intersections) and then check the emptiness of $\mathcal{A}'$. The next section explains how to do this. The size of the ata $\mathcal{A}$ is polynomial in the sizes of $\mathcal{M}_{\text{out}}$ and of $\mathcal{T}$. The size of $\mathcal{A}'$ is thus polynomial in the sizes of $\mathcal{M}_{\text{in}}$, $\mathcal{M}_{\text{out}}$, and $\mathcal{T}$.

3.2 Emptiness check

Let $\mathcal{A} = (\Xi, \Xi_0, \Phi)$ an alternating tree automaton. We want to decide whether the set $\mathcal{L}(\mathcal{A})$ is empty or not. We first define the following system of implications ρ where we introduce propositional variables $\overline{X}$ consisting of all subsets of Ξ:

$$\rho \;=\; \{\overline{X} \Leftarrow \overline{X}_1 \wedge \ldots \wedge \overline{X}_n \mid \;\; \exists a^{(n)}.\, (\overline{X}_1, \ldots, \overline{X}_n) \in \text{DNF}(\textstyle\bigwedge_{X \in \overline{X}} \Phi(X, a^{(n)}))\}$$

Here, $\text{DNF}(\phi)$ computes ϕ's disjunctive normal form by pushing intersections under unions and regrouping atoms of the form $\downarrow_i X$ for a fixed i; the result is formatted as a set of n-tuples of state sets. More precisely:

$$\begin{array}{lcll}
\text{DNF}(\top) & = & \{(\emptyset, \ldots, \emptyset)\} & \\
\text{DNF}(\bot) & = & \emptyset & \\
\text{DNF}(\phi_1 \wedge \phi_2) & = & \{(\overline{X}_1 \cup \overline{Y}_1, \ldots, \overline{X}_n \cup \overline{Y}_n) \mid (\overline{X}_1, \ldots, \overline{X}_n) \in \text{DNF}(\phi_1),\ (\overline{Y}_1, \ldots, \overline{Y}_n) \in \text{DNF}(\phi_2)\} & \\
\text{DNF}(\phi_1 \vee \phi_2) & = & \text{DNF}(\phi_1) \cup \text{DNF}(\phi_2) & \\
\text{DNF}(\downarrow_h X) & = & \{(\emptyset, \ldots, \emptyset, \{X\}, \emptyset, \ldots, \emptyset)\} & \text{(the } h\text{-th element is } \{X\}\text{)}
\end{array}$$

Then, with the system of implications above, we verify that $\rho \vdash \{X\}$ for some $X \in \Xi_0$. The judgment $\rho \vdash \overline{X}$ here is defined such that it holds when it can be derived by the single rule: if ρ contains $\overline{X} \Leftarrow \overline{X}_1 \wedge \ldots \wedge \overline{X}_n$ and $\rho \vdash \overline{X}_i$ for any $i = 1, \ldots, n$, then $\rho \vdash \overline{X}$.

Each propositional variable $\overline{X}$ intuitively denotes that the intersection of the sets denoted by all the states in $\overline{X}$ is non-empty: $\bigcap_{X \in \overline{X}} [\![X]\!] \neq \emptyset$. Thus, we can prove the following.

Proposition 1 *$\mathcal{L}(\mathcal{A}) \neq \emptyset$ iff $\rho \vdash \{X\}$ for some $X \in \Xi_0$.*

Proof: The result follows by showing that $v \in \bigcap_{X \in \overline{X}} [\![X]\!]$ for some v iff $\rho \vdash \overline{X}$. The "only if" direction can be proved by induction on the structure of v. The "if" direction can be proved by induction on the derivation of $\rho \vdash \overline{X}$. □

This emptiness check can be implemented in linear size with respect to the size of ρ, which itself is exponential in the size of $\mathcal{A}$.

4 Algorithm and optimizations

As we explained above, our algorithm splits the type-checking process in two phases: first, we compute an alternating tree automaton from the output type and the mtt; second, we check emptiness of this tree automaton. In this section, we are going to describe some details and optimizations about these two phases.

4.1 Backward inference

A simple algorithm to compute the input type as an alternating tree automaton is to follow naively the formal construction given in Section 3. A first observation is that it is possible to build the automaton lazily, starting from the initial states, producing new states and computing $\Phi(\_)$ only on demand. This is sometimes useful since the emptiness check algorithm we are going to describe in the next section works in a top-down way and will not always materialize the whole automaton.

The defining equations for the function Inf as given in Section 3 produce huge formulas. We will now describe new equations that produce much smaller formulas in practice. Before describing them, it is convenient to generalize the notation $\mathrm{Inf}(e, q, \vec{q})$ by allowing a *set of states* $\overline{q} \subseteq Q$ instead of a single state $q \in Q$ for the output type. Intuitively, we want $\mathrm{Inf}(e, \overline{q}, \vec{q})$ to be semantically equivalent to $\bigvee_{q \in \overline{q}} \mathrm{Inf}(e, q, \vec{q})$. We obtain a direct definition of $\mathrm{Inf}(e, \overline{q}, \vec{q})$ by adapting the rules for $\mathrm{Inf}(e, q, \vec{q})$:

$$
\begin{array}{lcl}
\mathrm{Inf}(b^{(m)}(e_1, \ldots, e_m), \overline{q}, \vec{q}) & = & \displaystyle\bigvee_{(q \leftarrow b^{(m)}(\vec{q'})) \in \Delta, q \in \overline{q}} \bigwedge_{j=1\ldots,m} \mathrm{Inf}(e_j, \{q'_j\}, \vec{q}) \\
\mathrm{Inf}(p^{(l)}(x_h, e_1, \ldots, e_l), \overline{q}, \vec{q}) & = & \displaystyle\bigvee_{\vec{q'} \in Q^l} \left(\downarrow_h \langle p^{(l)}, \overline{q}, \vec{q'} \rangle \wedge \bigwedge_{j=1,\ldots,l} \mathrm{Inf}(e_j, \{q'_j\}, \vec{q}) \right) \\
\mathrm{Inf}(y_j, \overline{q}, \vec{q}) & = & \begin{cases} \top & (q_j \in \overline{q}) \\ \bot & (q_j \notin \overline{q}) \end{cases}
\end{array}
$$

We have used the notation $\downarrow_h \langle p^{(l)}, \overline{q}, \vec{q'} \rangle$. Intuitively, this should be semantically equivalent to the union $\bigvee_{q \in \overline{q}} \downarrow_h \langle p^{(l)}, q, \vec{q'} \rangle$. Instead of using this as a definition, we prefer to change the set of states of the automaton:

$$
\begin{array}{lcl}
\Xi & = & \{\langle p^{(k)}, \overline{q}, q_1, \ldots, q_k \rangle \mid p^{(k)} \in P,\ \overline{q} \subseteq Q, \vec{q} \in Q^k\} \\
\Xi_0 & = & \{\langle p_0, Q_F \rangle \mid p_0 \in P_0\} \\
\Phi(\langle p^{(k)}, \overline{q}, \vec{q} \rangle, a^{(n)}) & = & \bigvee_{(p^{(k)}(a^{(n)}(\vec{x}), \vec{y}) = e) \in R} \mathrm{Inf}(e, \overline{q}, \vec{q}).
\end{array}
$$

In theory, this new alternating tree automaton could have exponentially many more states. However, in practice, and because of the optimizations we will describe now, this actually reduces significantly the number of states that need to be computed.

The sections below will use the semantical equivalence $\bigvee_{q \in \overline{q}} \mathrm{Inf}(e, \{q\}, \vec{q}) \simeq \mathrm{Inf}(e, \overline{q}, \vec{q})$ mentioned above in order to simplify formulas.

4.1.1 Cartesian factorization

The rule for the constructor expression $b^{(m)}(e_1, \ldots, e_m)$ can be written:

$$
\mathrm{Inf}(b^{(m)}(e_1, \ldots, e_m), \overline{q}, \vec{q}) = \bigvee_{\vec{q'} \in \Delta(\overline{q}, b^{(m)})} \bigwedge_{j=1\ldots,m} \mathrm{Inf}(e_j, \{q'_j\}, \vec{q})
$$

where $\Delta(\overline{q}, b^{(m)}) = \{\vec{q'} \mid q \leftarrow b^{(m)}(\vec{q'}) \in \Delta, q \in \overline{q}\} \subseteq Q^m$. Now assume that we have a decomposition of this set $\Delta(\overline{q}, b^{(m)})$ as a union of l Cartesian products:

$$\Delta(\overline{q}, b^{(m)}) = (\overline{q}^1_1 \times \ldots \times \overline{q}^1_m) \cup \ldots \cup (\overline{q}^l_1 \times \ldots \times \overline{q}^l_m)$$

where the $\overline{q}^i_j$ are sets of states. It is always possible to find such a decomposition: at worst, using only singletons for the $\overline{q}^i_j$, we will have as many terms in the union as m-tuples in $\Delta(\overline{q}, b^{(m)})$. But often, we can produce a decomposition with fewer terms in the union. Let us write $\mathrm{Cart}(\Delta(\overline{q}, b^{(m)})$ for such a decomposition (seen as a subset of $(2^Q)^m$). One can then use the following rule:

$$\mathrm{Inf}(b^{(m)}(e_1, \ldots, e_m), \overline{q}, \vec{q}) = \bigvee_{(\overline{q}_1, \ldots, \overline{q}_m) \in \mathrm{Cart}(\Delta(\overline{q}, b^{(m)}))} \bigwedge_{j=1,..,m} \mathrm{Inf}(e_j, \overline{q}_j, \vec{q})$$

4.1.2 State partitioning

Intuition The rule for procedure call enumerates all the possible states for the value of parameters of the called procedure. In its current form, this rule always produces a big union with $|Q|^l$ terms. However, it may be the case that we don't need fully precise information about the value of a parameter to do the backward type inference.

Let us illustrate that with a simple example. Assume that the called procedure $p^{(1)}$ has a single parameter y_1 and that it never does anything else with y_1 than copying it (that is, any rule for p whose right-hand side mentions y_1 is of the form $p^{(1)}(a^{(n)}(x_1, \ldots, x_n), y_1) = y_1$). Clearly, all the states $\langle p, \overline{q}, q'_1 \rangle$ with $q'_1 \in \overline{q}$ are equivalent, and similarly for all the states $\langle p, \overline{q}, q''_1 \rangle$ with $q''_1 \notin \overline{q}$. This is because whether the result of the procedure call will be or not in $\overline{q}$ only depends on the input tree (because there might be other rules whose right-hand side don't involve y_1 at all) and on whether the value for the parameter is itself in $\overline{q}$ or not. In particular, we don't know to know exactly in which state the accumulator is. So the rule for calling this procedure could just be:

$$\begin{aligned}
&\mathrm{Inf}(p(x_h, e_1), \overline{q}, \vec{q}) \\
&\quad = \bigvee_{q'_1 \in Q} \downarrow_h \langle p, \overline{q}, q'_1 \rangle \wedge \mathrm{Inf}(e_1, \{q'_1\}, \vec{q}) \\
&\quad = \left(\bigvee_{q'_1 \in \overline{q}} \downarrow_h \langle p, \overline{q}, q'_1 \rangle \wedge \mathrm{Inf}(e_1, \{q'_1\}, \vec{q}) \right) \cup \left(\bigvee_{q''_1 \in Q \setminus \overline{q}} \downarrow_h \langle p, \overline{q}, q''_1 \rangle \wedge \mathrm{Inf}(e_1, \{q''_1\}, \vec{q}) \right) \\
&\quad = \left(\downarrow_h \langle p, \overline{q}, q'_1 \rangle \wedge \mathrm{Inf}(e_1, \overline{q}, \vec{q}) \right) \vee \left(\downarrow_h \langle p, \overline{q}, q''_1 \rangle \wedge \mathrm{Inf}(e_1, Q \setminus \overline{q}, \vec{q}) \right)
\end{aligned}$$

where in the last line q'_1 (resp. q''_1) is chosen arbitrarily in $\overline{q}$ (resp. $Q \setminus \overline{q}$).

A new rule More generally, in the rule for a call to a procedure $p^{(l)}$, we don't need to consider all the l-tuples $\vec{q'}$, but only a subset of them that capture all the possible situations. First, we assume that for given procedure $p^{(l)}$ and output type $\overline{q}$, one can compute for each $j = 1, .., l$ an equivalence relation $E\langle p^{(l)}, \overline{q}, j \rangle$ such that:

$$(\forall j = 1, .., l.\ (q'_j, q''_j) \in E\langle p^{(l)}, \overline{q}, j \rangle) \Rightarrow \langle p^{(l)}, \overline{q}, \vec{q'} \rangle \simeq \langle p^{(l)}, \overline{q}, \vec{q''} \rangle \quad (*)$$

Let us look again at the right-hand side of the definition for $\mathrm{Inf}(p^{(l)}(x_h, e_1, \ldots, e_l), \overline{q}, \vec{q})$:

$$\mathrm{Inf}(p^{(l)}(x_h, e_1, \ldots, e_l), \overline{q}, \vec{q}) = \bigvee_{\vec{q'} \in Q^l} \left(\downarrow_h \langle p^{(l)}, \overline{q}, \vec{q'} \rangle \wedge \bigwedge_{j=1,\ldots,l} \mathrm{Inf}(e_j, \{q'_j\}, \vec{q}) \right)$$

Let us split this union according to the equivalence class of the q'_j modulo the relations $E\langle p^{(l)}, \overline{q}, j \rangle$. If for each j, we choose an equivalence class $\overline{q}_j$ for the relation $E\langle p^{(l)}, \overline{q}, j \rangle$ (we write $\overline{q}_j \lhd E\langle p^{(l)}, \overline{q}, j \rangle$), then all the states $\langle p^{(l)}, \overline{q}, \vec{q'} \rangle$ with $\vec{q'} \in \overline{q}_1 \times \ldots \times \overline{q}_l$ are equivalent to $\langle p^{(l)}, \overline{q}, \mathrm{C}(\overline{q}_1 \times \ldots \times \overline{q}_l) \rangle$, where C is a choice function (it picks an arbitrary element from its argument). We can thus rewrite the right hand-side to:

$$\bigvee_{\overline{q}_1 \lhd E\langle p^{(l)}, \overline{q}, 1 \rangle, \ldots, \overline{q}_l \lhd E\langle p^{(l)}, \overline{q}, l \rangle} \left(\downarrow_h \langle p^{(l)}, \overline{q}, \mathrm{C}(\overline{q}_1 \times \ldots \times \overline{q}_l) \rangle \wedge \bigvee_{\vec{q'} \in \overline{q}_1 \times \ldots \times \overline{q}_l} \bigwedge_{j=1,\ldots,l} \mathrm{Inf}(e_j, \{q'_j\}, \vec{q}) \right)$$

The union of all the formulas $\bigwedge_{j=1,..,l} \mathrm{Inf}(e_j, \{q'_j\}, \vec{q})$ for $\vec{q'} \in \overline{q}_1 \times \ldots \times \overline{q}_l$ is equivalent to $\bigwedge_{j=1,..,l} \mathrm{Inf}(e_j, \overline{q}_j, \vec{q})$. Consequently, we obtain the following new rule:

$$\mathrm{Inf}(p^{(l)}(x_h, e_1, \ldots, e_l), \overline{q}, \vec{q}) = \bigvee_{\overline{q}_1 \lhd E\langle p^{(l)}, \overline{q}, 1 \rangle, \ldots, \overline{q}_l \lhd E\langle p^{(l)}, \overline{q}, l \rangle} \left(\downarrow_h \langle p^{(l)}, \overline{q}, \mathrm{C}(\overline{q}_1 \times \ldots \times \overline{q}_l) \rangle \wedge \bigwedge_{j=1,\ldots,l} \mathrm{Inf}(e_j, \overline{q}_j, \vec{q}) \right)$$

In the worst case, all the equivalence relations $E\langle p^{(l)}, \overline{q}, j \rangle$ are the identity, and the right-hand side is the same as for the old rule. But if we can identify larger equivalence classes, we can significantly reduce the number of terms in the union on the right-hand side.

Computing the equivalence relations Now we will give an algorithm to compute the relations $E\langle p^{(k)}, \overline{q}, j \rangle$ satisfying the condition $(*)$. We will also define equivalence relations $E[e, \overline{q}, j]$ for any (n,k)-expression e (with $j = 1, .., k$), such that:

$$(\forall j = 1, .., k. (q'_j, q''_j) \in E[e, \overline{q}, j]) \Rightarrow \mathrm{Inf}(e, \overline{q}, \vec{q'}) \simeq \mathrm{Inf}(e, \overline{q}, \vec{q''})$$

We can use the rules used to define the formulas $\mathrm{Inf}(e, \overline{q}, \vec{q})$ in order to obtain sufficient conditions to be satisfied so that these properties hold. We will express these conditions by a system of equations. Before giving this system, we need to introduce some notations. If E_1 and E_2 are two equivalence relations on Q, we write $E_1 \sqsubseteq E_2$ if $E_2 \subseteq E_1$ (when equivalence relations are seen as subsets of Q^2). The smallest equivalence relation for this ordering is the equivalence relation with a single equivalence class. The largest equivalence relation is the identity on Q. For two equivalence relations E_1, E_2, we can define their least upper bound $E_1 \sqcup E_2$ as the set-theoretic intersection. For an equivalence relation E and a set of states $\overline{q}$, we write $\overline{q} \lhd E$ if $\overline{q}$ is one of the equivalence class modulo E. Abusing the notation by identifying an equivalence relation with the partition it induces on Q, we will write $\{Q\}$ for the smallest relation and $\{\overline{q}, Q \backslash \overline{q}\}$ for the relation with the two equivalence classes $\overline{q}$ and its complement. The system of equations is derived from the rules used to define the function Inf:

$$
\begin{array}{lcl}
E[b^{(m)}(e_1,\ldots,e_m),\overline{q},i] & \supseteq & \bigsqcup\{E[e_j,\overline{q}_j,i] \mid (\overline{q}_1,\ldots,\overline{q}_m) \in \mathrm{Cart}(\Delta(\overline{q},b^{(m)})),\ j=1..m\} \\
E[p^{(l)}(x_h,e_1,\ldots,e_l),\overline{q},i] & \supseteq & \bigsqcup\{E[e_j,\overline{q}_j,i] \mid \overline{q}_j \triangleleft E\langle p^{(l)},\overline{q},j\rangle,\ j=1..l\} \\
E[y_j,\overline{q},i] & \supseteq & \begin{cases} \{\overline{q},Q\backslash\overline{q}\} & (i=j) \\ \{Q\} & (i\neq j) \end{cases} \\
E\langle p^{(k)},\overline{q},j\rangle & \supseteq & \bigsqcup\{E[e,\overline{q},j] \mid p^{(k)}(a^{(n)}(\vec{x}),\vec{y})=e) \in R\}
\end{array}
$$

Let us explain why these conditions imply the required properties for the equivalence relation and how they are derived from the rules defining Inf. We will use an intuitive induction argument (on expressions), even though a formal proof actually requires an induction on trees. Consider the rule for the procedure call. The new rule we have obtained above implies that in order to have $\mathrm{Inf}(p^{(l)}(x_h,e_1,\ldots,e_l),\overline{q},\vec{q'}) \simeq \mathrm{Inf}(p^{(l)}(x_h,e_1,\ldots,e_l),\overline{q},\vec{q''})$, it is sufficient to have $\mathrm{Inf}(e_j,\overline{q}_j,\vec{q'}) \simeq \mathrm{Inf}(e_j,\overline{q}_j,\vec{q''})$ for all $j=1,..,l$ and for all $\overline{q}_j \triangleleft E\langle p^{(l)},\overline{q},j\rangle$, and thus, by induction, it is also sufficient to have $(q'_i,q''_i) \in E[e_j,\overline{q}_j,i]$ for all i, for all $j=1,..,l$ and for all $\overline{q}_j \triangleleft E\langle p^{(l)},\overline{q},j\rangle$. In other words, a sufficient condition is $(q'_i,q''_i) \in \bigcap\{E[e_j,\overline{q}_j,i] \mid \overline{q}_j \triangleleft E\langle p^{(l)},\overline{q},j\rangle,\ j=1..l\}$, from which we obtain the equation above (we recall that $\sqcup$ corresponds to set-theoretic intersection of relations). The reasoning is similar for the constructor expression. Indeed, the rule we have obtained in the previous section tells us that in order to have $\mathrm{Inf}(b^{(m)}(e_1,\ldots,e_m),\overline{q},\vec{q'}) \simeq \mathrm{Inf}(b^{(m)}(e_1,\ldots,e_m),\overline{q},\vec{q''})$, it is sufficient to have $\mathrm{Inf}(e_j,\overline{q}_j,\vec{q'}) \simeq \mathrm{Inf}(e_j,\overline{q}_j,\vec{q''})$ for all $(\overline{q}_1,\ldots,\overline{q}_m) \in \mathrm{Cart}(\Delta(\overline{q},b^{(m)}))$ and $j=1,..,m$.

As we explained before, it is desirable to compute equivalence relations with large equivalence classes (that is, small for the $\sqsubseteq$ ordering). Here is how we can compute a family of equivalence relations satisfying the system of equations above. First, we consider the CPO of functions mapping a triple $(e,\overline{q},i)$ to an equivalence relation on Q and we reformulate the system of equation as finding an element x of this CPO such that $f(x) \sqsubseteq x$, where f is obtained from the right-hand sides of the equations. To compute such an element, we start from x_0 the smallest element of the CPO, and we consider the sequence defined by $x_{n+1} = x_n \sqcup f(x_n)$. Since this sequence is monotonic and the CPO is finite, the sequence reaches a constant value after a finite number of iterations. This value x satisfies $f(x) \sqsubseteq x$ as expected. We conjecture that this element is actually a smallest fixpoint for f, but we have no proof of this fact (note that the function f is not monotonic).

4.1.3 Sharing the computation

Given the rules defining the formulas $\mathrm{Inf}(e,\overline{q},\vec{q})$, we might end up computing the same formula several times. A very classical optimization consists in memoizing the results of such computations. This is made even more effective by hash-consing the expressions. Indeed, in practice, for a given mtt procedure, many constructors have identical expressions.

4.1.4 Complementing the output

In the example at the beginning of the previous subsection, we have displayed a formula where both $\mathrm{Inf}(e,\overline{q},\vec{q})$ and $\mathrm{Inf}(e,Q\backslash\overline{q},\vec{q})$ appear. One may wonder what is the relation between these two sub-formulas. Let us recall the required properties for these two formulas:

$$[\![\mathrm{Inf}(e,\overline{q},\vec{q})]\!] = \{v \mid [\![p]\!](\vec{v},\vec{w}) \cap [\![\overline{q}]\!] \neq \emptyset\}$$

$$[\![\mathrm{Inf}(e,Q\backslash\overline{q},\vec{q})]\!] = \{v \mid [\![p]\!](\vec{v},\vec{w}) \cap [\![Q\backslash\overline{q}]\!] \neq \emptyset\}$$

(for $\vec{w} \in [\![\vec{q}]\!]$). Note that $[\![Q\backslash\vec{q}]\!]$ is the complement of $[\![\vec{q}]\!]$. As a consequence, if $[\![p]\!]$ is a total deterministic function (that is, if $[\![p]\!](\vec{v},\vec{w})$ is always a singleton), then $[\![\mathrm{Inf}(e, Q\backslash\overline{q}, \vec{q})]\!]$ is the complement of $[\![\mathrm{Inf}(e,\overline{q},\vec{q})]\!]$. If we extend the syntax of formula in alternating tree automata with negation (whose semantics is trivial to define), we can thus introduce the following rule:

$$\mathrm{Inf}(e,\overline{q},\vec{q}) = \neg\mathrm{Inf}(e, Q\backslash\overline{q},\vec{q})$$

to be applied e.g. when the cardinal of $\overline{q}$ is strictly larger than half the cardinal of Q. In practice, we observed a huge impact of this optimization: the number of constructed states is divided by two in all our experiences, and the emptiness algorithm runs much more efficiently. Also, because of the memoization technique mentioned above, this optimization allows us to share more computation. That said, we don't have a clear explanation for the very important impact of this optimization.

The rule above can only be applied when the expression e denotes a total and deterministic function. We use a very simple syntactic criterion to ensure that: we require all the reachable procedures $p^{(k)}$ to have exactly one rule $p^{(k)}(a^{(n)}(x_1,\ldots,x_n), y_1,\ldots,y_k) \to e$ for each symbol $a^{(n)}$.

4.2 Emptiness algorithm

In this section, we describe an efficient algorithm to check emptiness of an alternating tree automaton. Instead of giving directly the final version of the algorithm which would look quite obscure, we prefer to start describing formally a simple algorithm and then explain various optimizations.

Let $\mathcal{A} = (\Xi, \Xi_0, \Phi)$ be an ata as defined in Section 2.2. Negation (as introduced in Section 4.1.4) will be considered later when describing optimizations. The basic algorithm relies on a powerset construction to translate $\mathcal{A}$ into a bottom-up tree automaton $\mathcal{M} = (Q, Q_F, \Delta)$. We define Q as the powerset 2^{Ξ}. Intuitively, a state $\overline{X} = \{X_1,\ldots,X_m\}$ in Q represents the intersection of the ata states X_i. For such a state and a tag $a^{(n)}$, one must thus consider the formula $\varphi(\overline{X}, a^{(n)}) = \bigwedge_{i=1,..,m} \Phi(X_i, a^{(n)})$, and put in Δ transitions of the form $\overline{X} \leftarrow a^{(n)}(\overline{X}_1,\ldots,\overline{X}_n)$ to mimic the formula $\varphi(\overline{X}, a^{(n)})$. First, we put $\varphi(\overline{X}, a^{(n)})$ in disjunctive normal form, using the DNF function introduced in Section 2:

$$\varphi(\overline{X}, a^{(n)}) \simeq \bigvee_{(\overline{X}_1,\ldots,\overline{X}_n)\in \mathrm{DNF}(\varphi(\overline{X},a^{(n)}))} \bigwedge_{i=1,..,n} \bigwedge_{X\in\overline{X}_i} \downarrow_i X$$

The transition relation Δ consists of all the transitions $\overline{X} \leftarrow a^{(n)}(\overline{X}_1,\ldots,\overline{X}_n)$ such that $(\overline{X}_1,\ldots,\overline{X}_n) \in \mathrm{DNF}(\varphi(\overline{X}, a^{(n)}))$. One defines $Q_F = \{\{X\} \mid X \in \Xi_0\}$. One can easily establish that $[\![\overline{X}]\!]_{\mathcal{M}} = \bigcap_{X\in\overline{X}} [\![X]\!]_{\mathcal{A}}$ and thus that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{A})$.

It is well-known that deciding emptiness of a bottom-up tree automaton can be done in linear time. The classical algorithm to do so works in a bottom up way and thus requires to fully materialize the automaton (which is of exponential size compared to the original ata). However, the construction above produces the automaton in a top-down way: for a given state $\overline{X}$, the construction gives all the transitions of the form $\overline{X} \leftarrow \ldots$. We can exploit this fact to derive an algorithm that doesn't necessarily require the whole automaton $\mathcal{M}$ to be built. The algorithm is given below in pseudo-code. The function `empty` takes a state $\overline{X}$ and returns `true` if it is empty or `false` otherwise. The test is done under a number of assertions represented by two global variables `P`,`N` which stores sets of $\mathcal{M}$-states. The set stored in `P` (resp. `N`) represents

positive (resp. negative) emptiness assumptions: states which are assumed to be empty (resp. non-empty). When the state $\overline{X}$ under consideration is neither in `P` or `N`, it is first assumed to be empty (added to `P`). This assumption is then checked recursively by exploring all the incoming transitions (for all possible tags and all components of the disjunctive normal form corresponding to this tag) and if a contradiction is found, the set of positive assumptions is backtracked and $\overline{X}$ is added to the set of negative assumptions. This memoization-based scheme is standard for coinductive algorithms.

```
function empty (X̄)
  if X̄ ∈ P then return true
  if X̄ ∈ N then return false
  let P_saved = P in
  P ← P ∪ {X̄};
  foreach a^(n) ∈ Σ
    if not (empty_formula (φ(X̄, a^(n)))) then
      P ← P_saved
      N := N ∪ {X̄}
      return false
  return true

function empty_formula (φ)
  foreach (X̄_1, ..., X̄_n) ∈ DNF(φ)
    if not (empty_sub (X̄_1, ..., X̄_n)) then
      return false
  return true

function empty_sub (X̄_1, ..., X̄_n)
  foreach 1 ≤ i ≤ n
    if (empty X̄_i) then
      return true
  return false
```

This algorithm is not linear in the size of the automaton $\mathcal{M}$ because of the backtracking on `P`. This backtracking can be avoided (as described in [6], Chapter 7 or in [22]), but the technique is rather involved and would make the presentation of the optimizations quite obscure. Moreover, we have indeed implemented the non-backtracking version (with all the optimizations) but we did not observe any noticeable speedup in our tests.

A first optimization improves the effectiveness of the memoization sets `P` and `N`. It is based on the fact that if $\overline{X}_1 \subseteq \overline{X}_2$ then $[\![\overline{X}_2]\!] \subseteq [\![\overline{X}_1]\!]$. As a consequence, if $\overline{X}' \subseteq \overline{X}$ for some $\overline{X}' \in$ `P`, then `empty`$(\overline{X})$ can immediately return `true`. Similarly, if $\overline{X} \subseteq \overline{X}'$ for some $\overline{X}' \in$ `N`, then `empty`$(\overline{X})$ can immediately return `false`.

Enumeration and pruning of the disjunctive normal form The disjunctive normal form of a formula can be exponentially larger than the formula itself. Our first improvement consists in not materializing it but enumerating it lazily with a pruning technique that avoids the exponential behavior in many cases.

```
function empty_formula (φ)
```

```
 return (empty_dnf ([φ],(∅,...,∅)))
```

```
function empty_dnf (l,((X̄_1,...,X̄_n) as a)) =
 match l with
 | [] -> return false
 | ⊤ :: rest -> return (empty_dnf (rest,a))
 | ⊥ :: rest -> return true
 | φ_1 ∨ φ_2 :: rest ->
     if not (empty_dnf (φ_1 :: rest,a)) then return false
     return (empty_dnf (φ_2 :: rest,a))
 | φ_1 ∧ φ_2 :: rest ->
     return (empty_dnf (φ_1::φ_2::rest,a))
 | ↓_h X :: rest ->
     if empty (X̄_h ∪ {X})) then return true
     return (empty_dnf (rest,(X̄_1,...,X̄_h ∪ {X},...,X̄_n)))
```

The first argument of `empty_dnf` is a list of formula whose conjunction must be put in disjunctive normal form. The second argument is an n-tuple (where n is the arity of the current symbol) which accumulates a "prefix" of the current term of the disjunctive normal form being built. When an atomic formula $\downarrow_h X$ is found, the state X is added to the h-th component of the accumulator. Here we have included an important optimization: if the new state $\overline{X}_h \cup \{X\}$ denotes an empty set, then one can prune the enumeration. For instance, for a formula of the form $\downarrow_1 X \wedge \phi$ where X turns out to be empty, the enumeration will not even look at ϕ. This optimization enforces the invariant that no component of the accumulator denotes an empty set. As a consequence, when the function `empty_dnf` reaches an empty list of formulas, the accumulator represents an element of the disjunctive normal form for which `empty_sub` would return `false`.

The order in which we consider the two sub-formulas ϕ_1 and ϕ_2 in the formulas $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$ might have a big impact on performances. It might be worthwhile to look for heuristics guiding this choice.

Witness It is not difficult to see that the algorithm can be further instrumented in order to produce a witness for non-emptiness (that is, when `empty`$(\overline{X})$ returns `false`, it also returns a tree v which belongs to $[\![\overline{X}]\!]$). To do so, we keep for each state in `N` a witness, and we also attach a witness to each component of the accumulator $(\overline{X}_1, \ldots, \overline{X}_n)$ in the enumeration for the disjunctive normal form. When checking for the emptiness of $\overline{X}_h \cup \{X\}$, we know that $\overline{X}_h$ is a non-empty state, and we have at our disposal a witness v for this state. Before doing the recursive call to `empty`, we can first check whether this witness v is in $[\![X]\!]$ (this can be done very efficiently). If this is the case, we know that $\overline{X}_h \cup \{X\}$ is also non-empty. In practice, this optimization avoids many calls to `empty`.

Negation and reflexivity We have mentioned in Section 4.1.4 an optimization which introduces alternating formulas with negation. Using De Morgan's laws, we can push the negation down and thus assume that it can only appear immediately above an atomic formula $\downarrow_i X$. Of course, it is possible to get rid of the negation by introducing for each state X a dual state $\neg X$ whose transition formula (for each tag) is the negation of the one for X; this only doubles the number of states. However, we prefer to support directly in the algorithm negated atomic

formulas $\neg \downarrow_i X$, because we can use the very simple fact that it denotes a set which does not intersect $\downarrow_i X$. The algorithm is thus modified to work with pairs of sets of $\mathcal{A}$-states, written $(\overline{X}, \overline{Y})$, which intuitively represents the set $\bigcap_{X \in \overline{X}} [\![X]\!]_{\mathcal{A}} \setminus \bigcup_{Y \in \overline{Y}} [\![Y]\!]_{\mathcal{A}}$. We define $\varphi((\overline{X}, \overline{Y}), a^{(n)})$ as $\bigwedge_{X \in \overline{X}} \Phi(X, a^{(n)}) \wedge \bigwedge_{Y \in \overline{Y}} \neg\Phi(Y, a^{(n)})$. The fact mentioned above translates itself into a shortcut case in the `empty` function: if the input is $(\overline{X}, \overline{Y})$ with $\overline{X} \cap \overline{Y} \neq \emptyset$, then the result is `true` (meaning that $(\overline{X}, \overline{Y})$ trivially denotes an empty set of trees).

The interesting cases for enumeration of the normal form are:

```
| ↓h X :: rest ->
    if empty (X̄h ∪ {X})) then return true
    return (empty_dnf (rest,((X̄1,Ȳ1),...,(X̄h ∪ {X},Ȳh),...,(X̄n,Ȳn))))
| ¬ ↓h Y :: rest ->
    if empty (Ȳh ∪ {Y})) then return true
    return (empty_dnf (rest,((X̄1,Ȳ1),...,(X̄h,Ȳh ∪ {Y}),...,(X̄n,Ȳn))))
```

Preprocessing Note the following trivial facts: For a formula $\phi_1 \wedge \phi_2$ to be empty, it is sufficient to have ϕ_1 or ϕ_2 empty; for a formula $\phi_1 \vee \phi_2$ to be empty, it is sufficient to have ϕ_1 and ϕ_2 empty; for a formula $\downarrow_i X$ to be empty, it is sufficient to have all the formulas $\Phi(X, a^{(n)})$ empty; for a formula $\neg \downarrow_i X$ to be empty, it is sufficient to have all the formulas $\neg\Phi(X, a^{(n)})$ empty.

Using these sufficient conditions and a largest fixpoint computation, we get a sound and efficient approximation of emptiness for formulas (it returns `true` only if the formula is indeed empty, but it may also return `false` is this case). We use this approximate criterion to replace any subformula ϕ which is trivially empty with $\bot$ and any subformula ϕ such that $\neg\phi$ is trivially empty with $\top$ (and then apply Boolean tautologies to eliminate $\bot$ and $\top$ as arguments of $\vee$ or $\wedge$). In practice, this optimization is very effective in reducing the size and complexity of formulas involved in the real (exact) emptiness check.

5 Experiments

We have experimented on our typechecker with various XML transformations implemented as mtts. Although we did not try very big transformations, we did work with large input and output tree automata automatically generated from the XHTML DTD (without taking XML attributes into account). Note that because this DTD has many tags, the mtts actually have many transitions since they typically copy tags, which requires all constructors corresponding to these tags to be enumerated. They do not have too many procedures, though. The bottom-up deterministic automaton that we generated from the XHTML DTD has 35 states.

Table 1 gives the elapsed times spent in typechecking several transformations and the number of states of the inferred alternating tree automaton that have been materialized. The experiment was conducted on an Intel Pentium 4 processor 2.80Ghz, running Linux kernel 2.4.27, and the typechecking time includes the whole process (determinization of the output type, backward inference, intersection with the input type, emptiness check). The typechecker is implemented in and compiled by Objective Caml 3.09.3.

We also indicate the number of procedures in each mtt, the maximum number of parameters, and the minimum integer b, if any, such that the mtt is syntactically b-bounded copying. Intuitively, the integer b captures the maximum number of times the mtt traverses any node of the input tree. This notion has been introduced in [12] where the existence of b is shown to imply

the polynomiality of the algorithm described in that paper (see also Appendix A.2). Here, we observe that even unbounded-copying mtts can be typechecked efficiently.

Transformation:	(1)	(2)	(3)	(4)	(5)	(6)	(7)
# of procedures:	2	2	3	5	4	6	6
Max # of parameters:	1	1	1	1	2	2	2
Bounded copying:	1	1	2	∞	∞	2	1
Type-checking time (ms):	1057	1042	0373	0377	0337	0409	0410
# of states in the ata:	147	147	43	74	37	49	49

Table 1: Results of the experiments

Unless otherwise stated, transformations are checked to have type `XHTML`→`XHTML` (i.e., both input and output types are `XHTML`). Transformation (1) removes all the `<b>` tags, keeping their contents. Transformation (2) is a variant that drops the `<div>` tags instead. The typechecker detects that the latter doesn't have type `XHTML`→`XHTML` by producing a counter-example:

```
<html><head><title/></head><body><div/></body>
```

Indeed, removing the `<div>` element may produce a `<body>` element with an empty content, which is not valid in XHTML. Transformation (3) copies all the `<a>` elements (and their corresponding subtrees) into a new `<div>` element and prepends the `<div>` to the `<body>` element. Transformation (4) groups together adjacent `<b>` elements, concatenating their contents. Transformation (5) extracts from an XHTML document a tree of depth 2 which represents the conceptual nesting structure of `<h1>` and `<h2>` heading elements (note that, in XHTML, the structure among headings is flat). Transformation (6) builds a tree representing a table of contents for the top two levels of itemizations, giving section and subsection numbers to them (where the numbers are constructed as Peano numerals), and prepends the resulting tree to the `<body>` element. Transformation (7) is a variant that only returns the table of contents.

We have also translated some transformations (that can be expressed as mtts) used by Tozawa and Hagiya in [26] (namely `htmlcopy`, `inventory`, `pref2app`, `pref2html`, `prefcopy`). Our implementation takes between 2ms and 6ms to typecheck these mtts, except for `inventory` for which it takes 22 ms. Tozawa and Hagiya report performance between 5ms and 1000ms on a Pentium M 1.8 Ghz for the satisfiability check (which corresponds to our emptiness check and excludes the time taken by backward inference). Although these results indicate our advantages over them to some extent, since the numbers are too small and they have not undertaken experiments as big as ours, it is hard to draw a meaningful conclusion.

6 Conclusion and Future Work

We have presented an efficient typechecking algorithm for mtts based on the idea of using alternating tree automata for representing the preimage of the given mtt obtained from the backward type inference. This representation was useful for deriving optimization techniques on the backward inference phase such as state partitioning and Cartesian factorization, and was also effective for speeding up the subsequent emptiness check phase by exploiting Boolean equivalences among formulas. Our experimental results confirmed that our techniques allow us to typecheck small sizes of transformations with respect to the full XHTML type. Finally,

we have also made an exact connection to two known algorithms, a classical one and Maneth-Perst-Seidl's, the latter implying an important polynomial complexity under a bounded-copying restriction.

The present work is only the first step toward a truly practical typechecker for mtts. In the future, we will seek for further improvements that allow typechecking larger and more complicated transformations. In particular, transformations with upward axes can be obtained by compositions of mtts as proved in [11] and a capability to typecheck such compositions of mtts in a reasonable time will be important. We have some preliminary ideas for the improvement and plan to pursue them as a next step. In the end, we hope to be able to handle (at least a reasonably large subset of) XSLT.

References

[1] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, 2001.

[2] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13. Springer-Verlag, Aug. 1991.

[3] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 51–63, 2003.

[4] J. Engelfriet and S. Maneth. A comparison of pebble tree transducers with macro tree transducers. *Acta Informatica*, 39(9):613–698, 2003.

[5] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31(1):710–146, 1985.

[6] A. Frisch. *Théorie, conception et réalisation d'un langage de programmation adapté à XML.* PhD thesis, Universit Paris 7, 2004.

[7] H. Hosoya. Regular expression filters for XML. *Journal of Functional Programming*, 16(6):711–750, 2006. Short version appeared in Proceedings of Programming Technologies for XML (PLAN-X), pp.13–27, 2004.

[8] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003. Short version appeared in Proceedings of Third International Workshop on the Web and Databases (WebDB2000), volume 1997 of Lecture Notes in Computer Science, pp. 226–244, Springer-Verlag.

[9] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2004. Short version appeared in Proceedings of the International Conference on Functional Programming (ICFP), pp.11-22, 2000.

[10] X. Leroy, D. Doligez, J. Garrigue, J. Vouillon, and D. Rémy. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`, 1996.

[11] S. Maneth, T. Perst, A. Berlea, and H. Seidl. XML type checking with macro tree transducers. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 283–294, 2005.

[12] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *International Conference on Database Theory (ICDT)*, pages 254–268, 2007.

[13] W. Martens and F. Neven. Typechecking top-down uniform unranked tree transducers. In *Proceedings of International Conference on Database Theory*, pages 64–78, 2003.

[14] W. Martens and F. Neven. Frontiers of tractability for typechecking simple XML transformations. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 23–34, 2004.

[15] T. Milo and D. Suciu. Type inference for queries on semistructured data. In *Proceedings of Symposium on Principles of Database Systems*, pages 215–226, Philadelphia, May 1999.

[16] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, May 2000.

[17] A. Møller, M. Ø. Olesen, and M. I. Schwartzbach. Static validation of XSL Transformations. Technical Report RS-05-32, BRICS, October 2005. Draft, accepted for TOPLAS.

[18] M. Murata. Transformation of documents and schemas by patterns and contextual conditions. In *Principles of Document Processing '96*, volume 1293 of *Lecture Notes in Computer Science*, pages 153–169. Springer-Verlag, 1997.

[19] K. Nakano and S.-C. Mu. A pushdown machine for recursive XML processing. In *APLAS*, pages 340–356, 2006.

[20] T. Perst and H. Seidl. Macro forest transducers. *Information Processing Letters*, 89(3):141–149, 2004.

[21] G. Slutzki. Alternating tree automata. *Theoretical Computer Science*, 41:305–318, 1985.

[22] T. Suda and H. Hosoya. Non-backtracking top-down algorithm for checking tree automata containment. In *Proceedings of Conference on Implementation and Applications of Automata (CIAA)*, pages 83–92, 2005.

[23] A. Tozawa. Towards static type checking for XSLT. In *Proceedings of ACM Symposium on Document Engineering*, 2001.

[24] A. Tozawa. XML type checking using high-level tree transducer. In *Functional and Logic Programming (FLOPS)*, pages 81–96, 2006.

[25] A. Tozawa and M. Hagiya. XML schema containment checking based on semi-implicit techniques. In *8th International Conference on Implementation and Application of Automata*, volume 2759 of *Lecture Notes in Computer Science*, pages 213–225. Springer-Verlag, 2003.

[26] A. Tozawa and M. Hagiya. Efficient decision procedure for a logic for XML. unpublished manuscipt, 2004.

In this section, we compare our algorithm with two existing algorithms, the classical one based on function enumeration and the Maneth-Perst-Seidl algorithm.

A.1 Classical Algorithm

The classical algorithm presented here is known as a folklore. Variants can be found in the literature for deterministic mtts [4] and for macro forest transducers [20]. The algorithm takes a dbta $\mathcal{M} = (Q, Q_F, \Delta)$ and an mtt $\mathcal{T} = (P, P_0, \Pi)$ and builds a dbta $\mathcal{N}' = (D, D_F, \delta)$ where:

$$\begin{array}{lcl} D & = & \{\langle p^{(m)}, \vec{q}\rangle \mid p^{(m)} \in P,\ \vec{q} \in Q^m\} \to 2^Q \\ D_F & = & \{d \in D \mid p_0 \in P_0,\ d(\langle p_0\rangle) \cap Q_F \neq \emptyset\} \\ \delta & = & \{d \leftarrow a^{(n)}(\vec{d}) \mid d(\langle p^{(m)}, \vec{q}\rangle) = \bigcup_{(p^{(m)}(a^{(n)}(\vec{x}),\vec{y}) \to e) \in \Pi} \mathrm{DInf}(e, \vec{d}, \vec{q})\} \end{array}$$

Here, the function DInf is defined as follows.

$$\begin{array}{lcl} \mathrm{DInf}(b^{(m)}(e_1, \ldots, e_m), \vec{d}, \vec{q}) & = & \{q' \mid\ q' \leftarrow b^{(m)}(\vec{q'}) \in \Delta,\ q'_j \in \mathrm{DInf}(e_j, \vec{d}, \vec{q})\ \ \forall j = 1, \ldots, m\ \} \\ \mathrm{DInf}(p(x_h, e_1, \ldots, e_l), \vec{d}, \vec{q}) & = & \bigcup\{d_h(\langle p, \vec{q'}\rangle) \mid q'_i \in \mathrm{DInf}(e_i, \vec{d}, \vec{q}), i = 1, \ldots, l\} \\ \mathrm{DInf}(y_j, \vec{d}, \vec{q}) & = & \{q_j\} \end{array}$$

The constructed automaton $\mathcal{N}'$ has, as states, the set of all functions that map each pair of a procedure and parameter types to a set of states. Intuitively, each state d represents the set of trees v such that, given a procedure $p^{(m)}$ and states $\vec{q}$, the set of results of evaluating p with the tree v and parameters $\vec{w}$ of types $\vec{q}$ is exactly described by the states $d(\langle p, \vec{q}\rangle)$. Thus, the initial states D_F represent the set of trees v such that the set of results from evaluating an initial procedure p_0 with v contains a tree accepted by the given dbta $\mathcal{M}$.

The function DInf computes, from given expression e, states $\vec{d}$ from D, and states $\vec{q}$ from Q, the set of states that exactly describes the set of results of evaluating e with a tuple $\vec{v}$ of trees of types $\vec{d}$ and parameters of types $\vec{q}$. Then we can collect in δ transitions $d \leftarrow a^{(n)}(\vec{d})$ for all $a^{(n)}$ and all $\vec{d}$ such that d is computed for all $p^{(m)}$ and all $\vec{q}$ by using DInf with the expression on $p^{(m)}$'s each rule for the symbol $a^{(n)}$. By this intuition, each of the three cases for DInf can be understood as follows.

- The set of results of evaluating the constructor expression $b^{(m)}(e_1, \ldots, e_m)$ is described by the set of states $\vec{q'}$ that have a transition $q' \leftarrow b^{(m)}(\vec{q'}) \in \Delta$ such that each q'_i describes the results of evaluating the corresponding subexpression e_i.
- The set of results of evaluating the procedure call $p(x_h, e_1, \ldots, e_l)$ is the set of results of evaluating p with the h-th input tree v_h and parameters resulted from evaluating each e_i. This set can be obtained by collecting the results of applying the function d_h to p and $\vec{q'}$ where each q'_i is one of the states that describe the set of results of e_i.
- The set of results of evaluating the variable expression y_j is exactly described by its type q_j.

Thus, the intuition behind is rather different from our approach. Nevertheless, we can prove that the resulting automaton from the classical algorithm is isomorphic to the one obtained from our approach followed by determinization.

Determinization of an ata can be done as follows. From an ata $\mathcal{A} = (\Xi, \Xi_0, \Phi)$, we build a dbta $\mathcal{N} = (R, R_F, \Gamma)$ where

$$\begin{array}{lcl} R & = & 2^{\Xi} \\ R_F & = & \{r \in \Xi \mid r \cap \Xi_0 \neq \emptyset\} \\ \Gamma & = & \{r \leftarrow a^{(n)}(\vec{r}) \mid r = \{X \mid \vec{r} \vdash \Phi(X, a^{(n)})\}\}. \end{array}$$

Here, the judgment $\vec{r} \vdash \phi$ is defined inductively as follows.

- $\vec{r} \vdash \phi_1 \wedge \phi_2$ if $\vec{r} \vdash \phi_1$ and $\vec{r} \vdash \phi_2$.
- $\vec{r} \vdash \phi_1 \vee \phi_2$ if $\vec{r} \vdash \phi_1$ or $\vec{r} \vdash \phi_2$.
- $\vec{r} \vdash \top$.
- $\vec{r} \vdash \downarrow_i X$ if $X \in r_i$.

That is, $\vec{r} \vdash \phi$ intuitively means that ϕ holds by interpreting each $\downarrow_i X$ as "X is a member of the set r_i".

The intuition behind determinization of an ata is the same as that of a nondeterministic tree automaton. That is, each state r in $\mathcal{N}$ denotes the set of trees v that have type X for all members X of r and do not have type Y for all non-members Y of r.

$$[\![r]\!] = \bigcap_{X \in r} [\![X]\!] \setminus \bigcup_{Y \notin r} [\![Y]\!] \tag{6}$$

This implies that any tree cannot have type r and r' at the same time when $r \neq r'$. Thus, the states of the tree automaton $\mathcal{N}$ form a partition of all the trees, that is, $\mathcal{N}$ is complete and deterministic. From this, we can understand the equivalence between $\mathcal{A}$ and $\mathcal{N}$ since each final state in $\mathcal{N}$ contains an initial state in the original ata $\mathcal{A}$ and therefore the set of such final states forms a partition of the sets denoted by the initial states of $\mathcal{A}$. Then, by using the formula (6), the interpretation "X is contained in r_i" of $\downarrow_i X$ in the judgment $\vec{r} \vdash \phi$ implies that $[\![r_i]\!] \subseteq [\![X]\!]$. Here, we can see a parallelism between the intuition of the judgment $\vec{v} \vdash \phi$ (where $\downarrow_i X$ is interpreted "$v_i \in [\![X]\!]$") and that of $\vec{r} \vdash \phi$. Indeed, a key property to the proof below is: $\vec{v} \vdash \phi$ if and only if $\vec{r} \vdash \phi$ for some $\vec{r}$ such that $\vec{v} \in [\![\vec{r}]\!]$.

Proposition 2 *$\mathcal{A}$ and $\mathcal{N}$ are equivalent.*

PROOF: To prove the result, it suffices to show the following.

$$v \in [\![r]\!] \iff r = \{X \mid v \in [\![X]\!]\}. \tag{7}$$

(Note that this is a rewriting of the equation (6).) Indeed, this implies

$$\begin{array}{lcll} v \in \mathcal{L}(\mathcal{N}) & \iff & v \in [\![R_F]\!] & \\ & \overset{by(7)}{\iff} & \exists r.\ (r \cap \Xi_0 \neq \emptyset\ \wedge\ r = \{X \mid v \in [\![X]\!]\}) & \\ & \iff & \exists X \in \Xi_0.\ v \in [\![X]\!] & \\ & \iff & v \in \mathcal{L}(\mathcal{A}). & \end{array}$$

The proof proceeds by induction on the structure of v. To show (7), the following is sufficient

$$(\exists \vec{r}.\ \vec{v} \in [\![\vec{r}]\!]\ \ \wedge\ \ \vec{r} \vdash \phi) \iff \vec{v} \vdash \phi. \tag{8}$$

since this implies (7):

$$\begin{aligned}
a^{(n)}(\vec{v}) \in [\![r]\!] &\iff \exists (r \leftarrow a^{(n)}(\vec{r})) \in \Gamma.\ \vec{v} \in [\![\vec{r}]\!] \\
&\iff \exists \vec{r}.\ r = \{X \mid \vec{r} \vdash \Phi(X, a^{(n)})\} \ \wedge\ \vec{v} \in [\![\vec{r}]\!] \\
&\overset{by(8)}{\iff} r = \{X \mid \vec{v} \vdash \Phi(X, a^{(n)})\} \\
&\iff r = \{X \mid a^{(n)}(\vec{v}) \in [\![X]\!]\}.
\end{aligned}$$

The proof of (8) itself is done by induction on the structure of ϕ. The "only if" direction is straightforward. For the "if" direction, let $r_i = \{X \mid v_i \in [\![X]\!]\}$ for $i = 1, \ldots, n$. By the induction hypothesis, (7) gives $v_i \in [\![r_i]\!]$. The rest is case analysis on ϕ.

- Case $\phi = \bot$. This never arises.
- Case $\phi = \top$. This case trivially holds.
- Case $\phi = \downarrow_h X$. From $\vec{v} \vdash \phi$, we have $v_h \in [\![X]\!]$ and therefore $X \in r_h$ by the definition of r_h. This implies the result.
- Case $\phi = \phi_1 \wedge \phi_2$. By the induction hypothesis, $\vec{v} \in [\![\vec{r'}]\!]$ and $\vec{r'} \vdash \phi_1$ with $\vec{v} \in [\![\vec{r''}]\!]$ and $\vec{r''} \vdash \phi_2$ for some $\vec{r'}$ and $\vec{r''}$. Since $\mathcal{N}$ is deterministic, both $\vec{r'}$ and $\vec{r''}$ actually equal to $\vec{r}$. Hence the result follows.
- Case $\phi = \phi_1 \vee \phi_2$. Similar to the previous case. □

Proposition 3 *Let $\mathcal{N}$ be obtained by determinizing the ata from the last section. Then, $\mathcal{N}$ and $\mathcal{N}'$ are isomorphic.*

PROOF: Define the function β from D to R as follows:

$$\beta(d) = \{\langle p^{(m)}, q, \vec{q}\rangle \mid p^{(m)} \in P,\ \vec{q} \in Q^m,\ q \in d(\langle p, \vec{q}\rangle)\}$$

Clearly, β is bijective: $\beta^{-1}(r)(\langle p, \vec{q}\rangle) = \{q \mid \langle p^{(m)}, q, \vec{q}\rangle \in r\}$. It remains to show that β is an isomorphism between $\mathcal{N}$ and $\mathcal{N}'$, that is, (1) $\beta(D_F) = R_F$ and (2) $\beta(\delta(d)) = \Gamma(\beta(d))$ for each d.

The condition (1) clearly holds since $d(p_0) \cap Q_F \neq \emptyset$ iff $\langle p_0, q\rangle \in \beta(d)$ for some $q \in Q_F$. To prove (2), it suffices to show

$$q \in \mathrm{DInf}(e, \vec{d}, \vec{q}) \text{ iff } \beta(\vec{d}) \vdash \mathrm{Inf}(e, q, \vec{q}).$$

Here, $\beta(d_1, \ldots, d_k)$ stands for $(\beta(d_1), \ldots, \beta(d_k))$. The proof is by induction on the structure of e.

- Case $e = b^{(m)}(e_1, \ldots, e_m)$.

$$\begin{aligned}
q \in \mathrm{DInf}(e, \vec{d}, \vec{q}) &\iff \exists (q \leftarrow b^{(m)}(\vec{q'})) \in \Delta.\ \forall j.\ q'_j \in \mathrm{DInf}(e_j, \vec{d}, \vec{q}) \\
&\overset{\text{byI.H.}}{\iff} \exists (q \leftarrow b^{(m)}(\vec{q'})) \in \Delta.\ \forall j.\ \beta(\vec{d}) \vdash \mathrm{Inf}(e_j, q'_j, \vec{q}) \\
&\iff \beta(\vec{d}) \vdash \bigvee_{(q \leftarrow b^{(m)}(\vec{q'})) \in \Delta} \bigwedge_{j=1\ldots,m} \mathrm{Inf}(e_j, q'_j, \vec{q}) \\
&\iff \beta(\vec{d}) \vdash \mathrm{Inf}(e, q, \vec{q})
\end{aligned}$$

- Case $e = p(x_h, e_1, \ldots, e_l)$:

$$\begin{aligned}
q \in \mathrm{DInf}(e, \vec{d}, \vec{q}) &\iff \bigcup\{d_h(p, \vec{q'}) \mid q'_i \in \mathrm{DInf}(e_i, \vec{d}, \vec{q}), i = 1, \ldots, l\} \\
&\iff \exists \vec{q'}.\ q \in d_h(p, \vec{q'}) \text{ and } \forall i.\ q'_i \in \mathrm{DInf}(e_i, \vec{d}, \vec{q'}) \\
&\overset{\text{byI.H.}}{\iff} \exists \vec{q'}.\ \langle p, q, \vec{q'}\rangle \in \beta(d_h) \text{ and } \forall i.\ \beta(\vec{d}) \vdash \mathrm{Inf}(e_i, q, \vec{q'}) \\
&\iff \beta(\vec{d}) \vdash \bigvee_{\vec{q'} \in Q^l} \bigwedge_{i=1,\ldots,l} \mathrm{Inf}(e_i, q, \vec{q'}) \wedge \downarrow_i \langle p, q, \vec{q'}\rangle \\
&\iff \beta(\vec{d}) \vdash \mathrm{Inf}(e, q, \vec{q})
\end{aligned}$$

- Case $e = y_j$. First, $q \in \mathrm{DInf}(y_j, \vec{d}, \vec{q})$ iff $q = q_j$. If $q = q_j$, then $\mathrm{Inf}(e, q, \vec{q}) = \top$ and therefore the RHS holds. If $q \neq q_j$, then $\mathrm{Inf}(e, q, \vec{q}) = \bot$ and therefore the RHS does not hold. □

A.2 Maneth-Perst-Seidl Algorithm

First, for simplicity in comparing the two algorithms, following [12], we consider an mtt where the input type is already encoded into procedures. That is, instead of the original mtt $\mathcal{T}$, we take an mtt $\mathcal{T}'$ and a bta $\mathcal{M}_{\mathrm{in}}$ such that

$$\mathcal{T}'(v) = \begin{cases} \mathcal{T}(v) & (v \in \mathcal{L}(\mathcal{M}_{\mathrm{in}})) \\ \emptyset & (\text{otherwise}). \end{cases}$$

That is, $\mathcal{T}'$ behaves exactly the same as $\mathcal{T}$ for the inputs from $\mathcal{L}(\mathcal{M}_{\mathrm{in}})$ but returns no result for the other inputs. See [12] for a concrete construction. Having done this, we only need to check that $\{v \mid \mathcal{T}'(v) \cap \mathcal{L}(\mathcal{M}) \neq \emptyset\} = \emptyset$.

In Maneth-Perst-Seidl algorithm, we construct a new mtt $\mathcal{U}$ from $\mathcal{T}' = (P, P_0, \Pi)$ specialized to the output-type dbta $\mathcal{M} = (Q, Q_F, \Delta)$ such that $\mathcal{U}(v) = \mathcal{T}'(v) \cap \mathcal{L}(\mathcal{M})$ for any tree v. This can be done by constructing the mtt $\mathcal{U} = (S, S_0, \Omega)$ where

$$\begin{aligned}
S &= \{\langle p^{(m)}, q, \vec{q}\rangle^{(m)} \mid p^{(m)} \in P, q, \vec{q} \in Q^m\} \\
S_0 &= \{\langle p_0, q\rangle \mid p_0 \in P_0,\ q \in Q_F\} \\
\Omega &= \{\langle p^{(m)}, q, \vec{q}\rangle(a^{(n)}(\vec{x}), \vec{y}) \to e' \mid (p^{(m)}(a^{(n)}(\vec{x}), \vec{y}) \to e) \in \Pi,\ e' \in \mathrm{Spec}(e, q, \vec{q})\}.
\end{aligned}$$

Here, we define the function Spec as follows.

$$\begin{aligned}
\mathrm{Spec}(a(e_1, \ldots, e_n), q, \vec{q}) &= \{a(e'_1, \ldots, e'_n) \mid q \leftarrow a(q'_1, \ldots, q'_n) \in \Delta,\ \forall i.\ e'_i \in \mathrm{Spec}(e_i, q'_i, \vec{q})\} \\
\mathrm{Spec}(p(x_h, e_1, \ldots, e_l), q, \vec{q}) &= \{\langle p, q, \vec{q'}\rangle(x_h, e'_1, \ldots, e'_l) \mid \vec{q'} \in Q^l,\ \forall i.\ e'_i \in \mathrm{Spec}(e_i, q'_i, \vec{q})\} \\
\mathrm{Spec}(y_i, q, \vec{q}) &= \{y_i\}
\end{aligned}$$

Intuitively, each procedure $\langle p, q, \vec{q}\rangle$ in the new mtt $\mathcal{U}$ yields, for any input value v and for any parameters $\vec{w}$ of types $\vec{q}$, the same results as p but restricted to type q:

$$[\![\langle p^{(m)}, q, \vec{q}\rangle]\!](v, \vec{w}) = [\![p^{(m)}]\!](v, \vec{w}) \cap [\![q]\!]$$

Similarly, $\mathrm{Spec}(e, q, \vec{q})$ yields, for any input values $\vec{v}$ and for all parameters $\vec{w}$ of types $\vec{q}$, the same results as e but restricted to type q:

$$[\![\mathrm{Spec}(e, q, \vec{q})]\!](\vec{v}, \vec{w}) = [\![e]\!](\vec{v}, \vec{w}) \cap [\![q]\!]$$

After thus constructing the mtt $\mathcal{U}$, the remaining is to check that the translation of $\mathcal{U}$ is empty, i.e., $\mathcal{U}(v) = \emptyset$ for any value v. This can be done as follows. Define first the following system of implications ρ' where we introduce propositional variables $\overline{X}$ consisting of all subsets of S:

$$\begin{array}{rcl}\rho' & = & \{\overline{X} \Leftarrow \overline{X}_1 \wedge \ldots \wedge \overline{X}_n \mid \exists a^{(n)}.\ \exists e_1, \ldots, e_k.\ \forall s^{(m)} \in \overline{X}.\ \exists j.\ (s^{(m)}(a^{(n)}(\vec{x}), \vec{y}) \rightarrow e_j) \in \Omega, \\ & & \qquad \forall i = 1, \ldots, n.\ \overline{X}_i = \{s' \in S \mid \exists j = 1, \ldots, k.\ s'(x_i, \ldots) \text{ occurs in } e_j\}\}\end{array}$$

and then verify that $\rho' \vdash \{s\}$ for some $s \in S_0$. Intuitively, each propositional variable $\overline{X}$ denotes whether there is some input v from which any procedure in the set $\overline{X}$ translates to some value with some parameters:

$$\exists v.\ \forall s^{(m)} \in \overline{X}.\ \exists \vec{w}.\ [\![s^{(m)}]\!](v, \vec{w}) \neq \emptyset$$

Now, we can prove that the system of implications obtained from the MPS and the one from our algorithm are exactly the same. From this, we can directly carry over useful properties found for the MPS algorithm to our algorithm. In particular, our algorithm has the same polynomial time complexity under the restriction of a finitely bounded number of copying [12].

Proposition 4 *Given an input type that accepts all trees and the mtt $\mathcal{T}'$ defined above, let $\mathcal{A}$ and ρ be the ata and the system of implications obtained by the algorithm in Section 3. Let Ξ_0 be $\mathcal{A}$'s initial states. Then, (ρ, Ξ_0) and (ρ', S_0) are identical.*

Proof: Note that both ρ and ρ' consist of all variables $\overline{X}$ where $\overline{X}$ is from the set $P \times Q \times Q^m$. The result follows by showing $\overline{X} \Leftarrow \overline{X}_1 \wedge \ldots \wedge \overline{X}_n \in \rho$ iff $\overline{X} \Leftarrow \overline{X}_1 \wedge \ldots \wedge \overline{X}_n \in \rho'$. It suffices to show for any $\overline{X}$ and i,

$$\exists e_1, \ldots, e_k.\ \forall s \in \overline{X}.\ \exists j.\ (s(a(\vec{x}), \vec{y}) \rightarrow e_j) \in \Omega, \overline{X}_i = \{s' \in S \mid \exists j = 1, \ldots, k.\ s'(x_i, \ldots) \text{ occurs in } e_j\}$$

iff

$$(\overline{X}_1, \ldots, \overline{X}_n) \in \mathrm{DNF}(\bigwedge_{s \in \overline{X}} \Phi(s, a)).$$

This follows by showing that, for all $(\overline{X}_1, \ldots, \overline{X}_n) \in \mathrm{DNF}(\mathrm{Inf}(e_1, q_1, \vec{q}_1) \wedge \ldots \wedge \mathrm{Inf}(e_k, q_k, \vec{q}_k))$,

$$\exists j = 1, \ldots, k.\ s'(x_i) \text{ occurs in } \mathrm{Spec}(e_j, q_j, \vec{q}_j) \iff s' \in \overline{X}_i.$$

This can be proved by induction on $|e_1| + \ldots + |e_k|$ where $|e|$ is the size of e. □

Corollary 1 *For any b-bounded copying mtt, our algorithm runs in polynomial time.*

B Alternating tree automata with bounded traversing

The corollary in the last section depends on the proof of polynomiality from [12]. It gives the information that the emptiness check for alternating automata has polynomial time complexity when the alternating automata is obtained by the basic backward inference algorithm from Section 3 when applied to a b-bounded copying mtt. It seems natural to look for a counterpart of the notion of b-bounded copying for alternating automata that directly ensures the polynomiality of the emptiness check.

Let $\mathcal{A} = (\Xi, \Xi_0, \Phi)$ be an ata. For each state $X \in \Xi$, we define the maximal traversal number $b[X]$ as the least fixpoint of a constraint system over $\mathcal{N} = \{1 < 2 < \ldots < \infty\}$, the complete

lattice of naturals extended with ∞. The constraint system consists of all the constraints of the form:

$$b[X] \geq b_i[\Phi(X, a^{(n)})]$$

for $a^{(n)} \in \Sigma$ and $1 \leq i \leq n$, where $b_i[\phi]$ is defined inductively:

$$\begin{array}{lcl} b_i[\top] & = & 0 \\ b_i[\bot] & = & 0 \\ b_i[\phi_1 \wedge \phi_2] & = & b_i[\phi_1] + b_i[\phi_2] \\ b_i[\phi_1 \vee \phi_2] & = & \max(b_i[\phi_1], b_i[\phi_2]) \\ b_i[\downarrow_h X] & = & \begin{cases} b[X] \text{ if } i = h \\ 0 \text{ if } i \neq h \end{cases} \end{array}$$

The ata $\mathcal{A}$ is (syntactically) b-bounded traversing if $b[X] \leq b$ for all $X \in X_0$.

We mention without proving it formally that when we apply our backward inference algorithm to a b-bounded copying mtt, then the resulting ata is b-bounded traversing. More precisely, we can show that $b[\langle p^{(k)}, q, \vec{q} \rangle] \leq b[p^{(k)}]$ where $b[p^{(k)}]$ denotes the maximal copy number for the procedure $p^{(k)}$, as defined in [12]. As a matter of fact, the optimizations given in Section 4.1 preserve this property (but the ata formally has exponentially many more states, even if in practice only a fraction of them is going to be materialized).

Now it remains to establish that the emptiness check for a b-bounded traversing ata runs in polynomial time. We define $b[\overline{X}]$ as $\Sigma_{X \in \overline{X}} b[X]$. For any b-formula ϕ and $(\overline{X}_1, \ldots, \overline{X}_n) \in \mathrm{DNF}(\phi)$ and $1 \leq i \leq n$, we observe that $b[\overline{X}_i] \leq b_i[\phi]$. The proof is by induction on the structure of ϕ. As a consequence, for any $(\overline{X}_1, \ldots, \overline{X}_n) \in \mathrm{DNF}(\bigwedge_{X \in \overline{X}} \Phi(X, a^{(n)}))$, we have $b[\overline{X}_i] \leq b[\overline{X}]$. So, if the ata is b-bounded traversing, then the emptiness check algorithm will only consider set of states $\overline{X}$ such that $b[\overline{X}] \leq b$. Since $b[\overline{X}]$ is a lower bound for the cardinal of $\overline{X}$ (because $b[X] \geq 1$ for all X), we see that the algorithm only looks at a polynomial number of set of states $\overline{X}$.

To conclude this section, we observe that the intersection of a b-bounded traversal ata and a b'-bounded traversal ata is a $(b + b')$-bounded traversal ata, and that a non-deterministic tree automaton is isomorphic to a 1-bounded traversal ata. This is useful to typecheck a b-bounded copying mtt, because we need to compute the intersection of the inferred ata, which is b-bounded traversal, and of the input type, which is given by a non-deterministic tree automaton. As a result, we obtain a $(b + 1)$-bounded ata.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399